John von Neumann Institute for Computing

Comparing Intel Thread Checker and Sun Thread Analyzer

Christian Terboven

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 669-676, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Comparing Intel Thread Checker and Sun Thread Analyzer

Christian Terboven

Center for Computing and Communication
RWTH Aachen University, 52074 Aachen, Germany
E-mail: terboven@rz.rwth-aachen.de

Abstract

Multiprocessor compute servers have been available for many years now. It is expected that the number of cores and threads per processor chip will increase in the future. Hence, parallel programming will become more common. Posix-/Win32-Threads and OpenMP are the most wide-spread programming paradigms for shared-memory parallelization.

At the first sight, programming for Posix-Threads or OpenMP may seem to be easily understandable. But for non-trivial applications, reasoning about the correctness of a parallel program is much harder than of a sequential control flow. The typical programming errors of shared-memory parallelization are data races and deadlocks. Data races cause the result of a computation to be non-deterministic and dependent on the timing of other events. In case of a deadlock two or more threads are waiting for each other. Finding those errors with traditional debuggers is hard, if not impossible.

This paper compares two software tools: Intel Thread Checker and Sun Thread Analyzer, that help the programmer in finding these errors. Experiences using both tools on multithreaded applications will be presented together with findings on the strengths and limitations of each product.

1 Introduction

Posix-Threads[1] and OpenMP[2] are the most popular programming paradigms for shared-memory parallelization. Typically it turns out that for non-trivial parallel applications reasoning about the correctness is much harder than for sequential control flow. The programming errors introduced by shared-memory parallelization are data races and deadlocks. Finding those errors with traditional debuggers is hard, as the errors are induced by the program flow, which itself is influenced by the debugger. In order to find problems incurred multithreading in acceptable time, we recommend to use specialized tools.

The first commercial product for detecting threading errors in Posix-Threads and OpenMP programs was KAI Assure for Threads. In 2000, Intel acquired Kuck and Associates, hence the Intel Thread Checker is the descendant of Assure. It is available on Intel and compatible architectures on Linux and Windows, the current version is 3.1.

With the release of Sun Studio 12 in May 2007, Sun released the Sun Thread Analyzer, although it has been available in the Studio Express program earlier. It is available on Intel and compatible and UltraSPARC architectures on Linux and Solaris.

Both programs support Posix-Threads and OpenMP programs, in addition Intel supports WIN32-Threads and Sun supports Solaris-threads.

This paper is organized as follows: In chapter 2 we take a look at how these two tools function and what kind of errors the user can expect to be detected. In chapter 3 we describe and compare our experiences of using these tools on small and larger software projects, with a focus on OpenMP programs. In chapter 4 we draw our conclusions.

2 Functioning and Usage

Both tools aim to detect data races and deadlocks in multithreaded programs. The simplest condition for a data race to exist is when all following requirements can occur concurrently (see[3] for a formal definition): Two or more threads of a single process access the same memory location concurrently, between two synchronization points in an OpenMP program, at least one of the threads modifies that location and the accesses to the location are not protected e.g. by locks or critical regions. Data races typically occur in a non-deterministic fashion, for example in an OpenMP program the sequence of the execution of parallel loop iterations is non-deterministic and may change from run to run. In the case of OpenMP, data races typically are caused by missing private clauses or missing synchronization constructs like barriers and critical regions. If threads wait for an event that will never happen, a deadlock occurs. Both kinds of errors do not happen in sequential programs.

Functioning. The basic functioning is pretty similar: Both tools require some sort of application instrumentation and both tools trace references to the memory, thread management operations (such as thread creation) and synchronization operations during an application run. With this information the definition of a data race (similar to the one given above) is checked for pairs of events on different threads. The results are post-processed and presented to the user via a GUI program. For each error report two stack traces (one for each thread) are shown. The deadlock detection works on the same trace data.

The dynamic analysis model implies the principle limitation of both tools[4]: Only problems in the program parts that have been executed during the analysis can be found. That means the analysis result depends on the actual input dataset. Therefore it is crucial to carefully select datasets for the analysis that cover all relevant code parts.

Usage. In order to discuss the usage of both tools, Fig. 1 shows a C version of the Jacobi solver from the OpenMP website, in which we deliberately introduced two parallelization mistakes. The *parallel region* spans the whole block. This *for*-loop is parallelized using OpenMP's worksharing (line 4), the loop iterations are grouped into chunks and then distributed among the threads. The loop variable j is private by default, that means it's name provides access to a different block of storage for each thread; loop variable i has been declared private in line 1. All other variables are shared by default, that means all threads access the same block of storage. The loop variables have to be private as they have different values for different threads during program runtime. The array U has to be shared, as two different threads will always access distinct parts of it.

The problems are related to $resid$ and $error$, which both are shared, thus data races will occur if the program is executed in parallel. $resid$ has to be private as it just stores a temporal computation result, $error$ has to be made a reduction variable in order to accumulate the results contributed by all threads. In this case, the minimum requirement on a data race detection tool is to report the races on $resid$ and $error$. In addition it would be very valuable if the user is pointed to the fact that $error$ depends on the result of individual

thread contributions and therefore it would not be correct to declare the variable private.

```
1   #pragma omp parallel private(i)
2   {
3      /* compute stencil, residual and update */
4   #pragma omp for
5      for (j=1; j<m-1; j++)
6         for (i=1; i<n-1; i++){
7            resid =(ax * (UOLD(j,i-1) + UOLD(j,i+1))
8                  + ay * (UOLD(j-1,i) + UOLD(j+1,i))
9                  + b * UOLD(j,i) - F(j,i) ) / b;
10           U(j,i) = UOLD(j,i) - omega * resid;
11           error = error + resid*resid;
12        }
13  }
14 /* end of parallel region */
```

Figure 1. Detail of the Jacobi program, erroneous OpenMP parallelization.

3 Comparison

In this chapter we compare the Intel Thread Checker and the Sun Thread Analyzer on different applications and scenarios. In subsections 3.1 and 3.2 we take a look at how the tools support the user in principle and discuss the simple OpenMP program presented above. In 3.3 we present how we used this kind of tools to do the actual parallelization. Subsection 3.4 examines how the tools handle C++ programs. Subsection 3.5 presents the memory consumption and runtime increase for selected applications. In subsection 3.6 we compare the ability to check libraries for thread safety. Subsection 3.7 briefly summarizes additional features offered by the tools.

3.1 Simple Use Case: Intel Thread Checker

The Intel Thread Checker supports two different analysis modes, which also can be combined. *Thread-count independent mode* allows for checking existing applications without the need of recompilation. Source information on reported problems can only be given if debug information is available. It requires the application to be executed under the control of the Thread Checker program. *Thread-count dependent mode* can provide additional symbolic information and it allows the code to be analyzed to be executed outside the control of the Thread Checker program. This can be a requirement for software components executed on demand. This mode requires the program to be compiled with the Intel compilers and the $-tcheck$ switch to be used. It allows additional analysis on OpenMP programs, but it is only applicable if the program flow does not depend on the number of threads used.

3.1.1 Thread-count independent mode

In the Jacobian example program, the most important part with respect to the parallelization has been shown in Fig. 1. The result of the analysis depends on the number of threads

used; when running with only one thread no data races are reported. In total 10 errors for 3 different program locations are reported.

A data race in line 8 is reported, both for unsynchronized write accesses by two threads and unsynchronized read and write access by two threads. Although no variable name is given, two source locations are displayed for each error and it is easy to recognize where the write and read accesses are happening. A data race in line 12 is reported, this report results from the race in variable $resid$, as that variable is read in this line. A data race in line 13 is reported, both for unsynchronized write accesses by two threads and unsynchronized read and write access by two threads, and one additional report depending on $resid$ as well.

Together with a detailed error description several guidelines on how to address the errors found are given via the GUI. For all three items it is proposed to either make the variables $private$ or synchronize the access to them. Of course, adding synchronization constructs does not make sense here, although privatizing the variable $error$ will not lead to the desired result as well. Nevertheless, we followed the Thread Checker's advice and rerun the application, again with binary instrumentation. This time, no error in the program is detected. While it is true that all data races were eliminated by privatizing the variables $resid$ and $error$, privatizing $error$ breaks the serial equivalence. That means, the output of the parallel program differs from the original program's output; in this case the result is wrong. As already mentioned above, the variable $error$ has to be made a reduction variable.

3.1.2 Thread-count dependent mode

This mode provides some advantages for OpenMP programs, if the program flow does not depend on the actual number of threads. In total only 5 errors for 2 different program locations are reported, this time the actual variable names $resid$ and $error$ are given. Again it is proposed to make $resid$ a private variable. For variable $error$ the data race occurred in three different kinds: unsynchronized write accesses by two threads and unsynchronized read and write access by two threads (in two different orders). For two of these three reports it is proposed to privatize the variable, but for one the user should consider declaring the variable as a reduction. The exact variable names are given, not just source code locations, and as the tool has additional knowledge of OpenMP, the advice given to the user respects the context and aids the user in correcting the data races.

If one ignores the advice, declares the variable private and reruns the analysis with source instrumentation, the Thread Checker even tells the user that the variable cannot be private, as this leads to undefined accesses in the parallel region. Unfortunately, then there is no further detailed advice given of how to overcome this situation.

3.2 Simple Use Case: Sun Thread Analyzer

In order to analyze an application with the Sun Thread Analyzer, it has to be recompiled using the Sun compilers using the switch *-xinstrument=datarace*. In addition, debug information generation has to be turned on, otherwise no useful source code locations can be given.

The result of the analysis depends on the number of threads used. When running the instrumented Jacobian solver with just one thread, no data races are reported. In order

to find any data races, the program to be analyzed has to be executed with two or more threads. Running with two threads, in total 6 data races for 2 different program locations are reported, namely for variables $resid$ and $error$. After privatizing these two variables, no data races are reported anymore, but again the program is not correct, as $error$ should be a reduction variable. The Sun Thread Analyzer does not give the user any special advice with respect to OpenMP.

3.3 Guidance in the Parallelization Process

Although these tools were designed to find errors in multithreaded programs as demonstrated above, they can assist during the actual parallelization process as well. After performance-critical hotspots were identified, the user can insert e.g. OpenMP parallelization directives, without reasoning about the correctness. Running an erroneous parallelization in one of the tools will show all code locations were data races occur. Then the user is responsible to correctly eliminate all occurrences, with more or less guidance by the tools. This process is iterated until all data races are resolved.

We successfully applied this technique on a FORTRAN code named Panta, beside others. As this code has been tuned for vector computers, the compute-intensive routines contained up to several hundred different scalar variables. Finding all variables that have to be privatized is a lot of work and also error-prone[5], nevertheless in OpenMP every variable has to be given a scope, either explicitly or implicitly. At that time we used the Assure tool to generate a list of all variables on which data races occurred. While most of these variables have to be made private, some have to be declared as first- or lastprivate or reduction. If the requirements for Intel Thread Checker's source instrumentation are fulfilled, the user is helped with these decisions as well.

A study on frequently made mistakes in OpenMP programs has been presented in[6]. While we agree that providing a checklist is a valid approach for the errors classified as performance-related, we recommend using the tools discussed here as early in the parallelization process as possible. From the mistakes classified as correctness-related, eight out of ten can be identified automatically by using the appropriate tools and compilers.

3.4 Handling of C++ Programs

Many of today's tools still have problems with C++ codes, e.g. profilers sometimes are unable to map measurement results to the source correctly. In order to investigate how good the Intel Thread Checker and Sun Thread Analyzer deal with C++ codes, we examined a CG solver that has been parallelized in OpenMP using the external parallelization approach[7]. That means, the parallel region spans the whole iteration loop, the OpenMP worksharing constructs are hidden inside member functions of the data types representing matrix and vector.

We found that both tools detect the data races we inserted into the C++ code. In addition, both tools reported the races where they occur and the user is able to use the GUI to navigate along the call stack. Nevertheless, the Intel Thread Checker with binary instrumentation and also the Sun Thread Analyzer reported additional data races to occur in the STL code, probably induced by the actual races. We found that, for example, when multiple threads create instances of objects and assign them to a single (shared) pointer,

beside the race in the pointer assignment additional races in the object type's constructor are reported. As the number of additional reports is significantly higher than the number of issues of real interest, the user might get distracted and it might take some time to understand the problem cause. Only the Intel Thread Checker in thread-count dependent mode shows just the occurring races.

3.5 Memory Consumption and Runtime

The memory consumption and the execution time of programs can increase dramatically during the analysis, as shown in Table 1 for selected programs. All experiments with the Intel Thread Checker were carried out on a Dell PowerEdge 1950 with two Intel Xeon CPUs, 8 GB of memory, running Scientific Linux 4.4, using the Intel 10.0 compilers and Threading Tools 3.1 update2. All experiments with the Sun Thread Analyzer were carried out on a Sun Fire V40z with four AMD Opteron CPUs, 8 GB of memory, running Solaris 10, using the Sun Studio 12 compilers. *Jacobi* denotes the Jacobian solver as discussed above. *SMXV* is a sparse Matrix-Vector multiplication written in C++, with a relatively small matrix size. *AIC* is an adaptive integration solver employing Nested OpenMP. In *SMXV* and *AIC* the program flow depends on the number of threads used and therefore the thread-count dependent mode of the Intel Thread Checker is not available. Both product

Table 1. Memory consumption (in MByte) and Performance / Runtime of selected programs.

Program	Jacobi		SMXV		AIC	
	Mem	MFLOP/s	Mem	MFLOP/s	Mem	Time
Original, Intel with 2 threads	5	621	40	929	4	5.0 s
Intel Thread Checker tc. indep., 2thr.	115	0.9	1832	3.5	30	9.5 s
Intel Thread Checker tc. dependent	115	3.1	—	—	—	—
Original, Sun with 2 threads	5	600	50	550	2	8.4 s
Sun Thread Analyzer with 2 threads	125	1.1	2020	0.8	17	8.5 s

documentations advise to use the smallest possible and still meaningful dataset. *Small* means that the memory consumption during normal runtime should be minimal, and the runtime itself should be as short as possible. This can be achieved by decreasing the grid resolution, limiting the number of iterations in a solver method, simulating just a couple of time steps, and so on. But it is important that the critical code paths are still executed. In order to ensure this, we found code coverage tools useful, which are provided by both vendors.

Our experience in practice is that it is impossible to analyze programs with production datasets. Looking at the very small Jacobian solver with a memory footprint of about 5 MB, it increases to significantly more than 100 MB with both tools, this is a factor of

about 25. The increase depends[8] on the parallelization of the program and we found it very hard to predict, but average factors of 10 to 20 forbid the usage of datasets using 10 GB of memory, for example, on most current machines.

The Intel Thread Checker in thread-count dependent mode and the Sun Thread Analyzer both run in parallel during the analysis, but we only found the Sun tool to profit from multiple threads. For example the SMXV program achieved with 2 threads 0.8 MFLOP/s, and 1.4 MFLOP/s with 4 threads.

3.6 Checking Libraries for Thread Safety

If library routines are called from possibly multiple threads at a time, the called routines have to be thread safe. Some advice given in the past in order to improve performance, such as declaring variables static, can lead to severe problems in multithreaded programs. We found many libraries in the public domain and also commercial ones to cause problems with parallel applications, for example techniques as reference-counting via internal static variables are still prevalent. Manually verifying library routines can be a tedious task, if not impossible, e.g. if the source code is not available.

```
1  #pragma omp parallel sections
2  {
3  #pragma omp section
4     routine1(&data1);
5  #pragma omp section
6     routine1(&data2);
7  #pragma omp section
8     routine2(&data3);
9  }
```

Figure 2. Pattern to check libraries for thread safety.

The pattern in Fig. 2 tests two scenarios: The thread safety of $routine1$ when two threads are calling this routine in parallel, and the thread safety of routines $routine1$ and $routine2$ being called concurrently. As the Thread Checker supports binary instrumentation without recompilation, it can be used to verify existing libraries for which no source code is available. Using the Sun Thread Analyzer, the source code of the library has to be present.

3.7 Other Features

Both tools offer the ability to detect deadlocks. If two threads already holding locks are requesting new locks such that a chain is formed, the application may deadlock depending on the thread scheduling. It is possible to detect both occurring deadlocks and potential deadlocks.

Some applications have been found to use explicit memory flushes for synchronization, instead of locks or critical regions. Doing so is not recognized by either of the tools, therefore data races are reported that will never occur during normal program runs. Both tools offer APIs to depict user-written synchronization mechanisms in order to avoid false positives.

4 Conclusion

We state that a user should never put a multithreaded program in production before using one of these tools. Both tools are capable of detecting data races in complex applications and using the provided GUIs, the user is presented with two call stacks to locate the races.

For OpenMP programs, the thread-count dependent mode of the Intel Thread Checker tool can provide a noticeable surplus value, but it is only available for a limited class of applications. The second advantage of the Intel Thread Checker is the ability to analyze existing binaries without the need of recompilation. As the runtime increase during the analysis is significantly, the Sun Thread Analyzer's ability to still offer some scalability is advantageous.

Independent of the tool used, the memory consumption may increase dramatically and finally render the tools unusable. Nevertheless, if suitable datasets are available, both tools can easily be embedded in the software development process.

References

1. IEEE, Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API), IEEE Std 1003.
2. OpenMP Architecture Review Board, *OpenMP Application Program Interface*, Version 2.5., (2005),
3. U. Banerjee, B. Bliss, Z. Ma and P. Petersen, *A theory of data race detection*, in: Proc. 2006 workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2006), (2006).
4. U. Banerjee, B. Bliss, Z. Ma and P. Petersen, *Unraveling data race detection in the Intel Thread Checker*, in: Proc. First Workshop on Software Tools for Multi-Core Systems (STMCS 2006), (2006).
5. Y. Lin, C. Terboven, D. an Mey and N. Copty, *Automatic scoping of variables in parallel regions of an OpenMP program*, in: Workshop on OpenMP Applications and Tools (WOMPAT 2004), Houston, USA, (2004).
6. M. Suess and C. Leopold, *Common mistakes in OpenMP and how to avoid them*, in: Second Internation Workshop on OpenMP (IWOMP 2005), (2005).
7. C. Terboven and D. an Mey, *OpenMP and C++*, in: Second International Workshop on OpenMP (IWOMP 2006), Reims, France, (2006).
8. P. Sack, B. E. Bliss, Z. Ma, P. Petersen and J. Torrellas, *Accurate and efficient filtering for the Intel Thread Checker race detector*, in: Proc. 1st workshop on Architectural and System Support for improving software dependability (ASID 2006), New York, USA, pp. 34–41, (2006).